LangGraphによる人間参加型（Human-in-the-Loop）AIエージェントシステム

著者: Masahiro Aoki
ドキュメントID：MT2025-AI-01-002
ORCID ID：0009-0007-9222-4181
所属: Moonlight Technologies 株式会社

文書バージョン	作成日	作成者	概要
Ver 1.0	2025年7月27日	Masahiro Aoki	初版

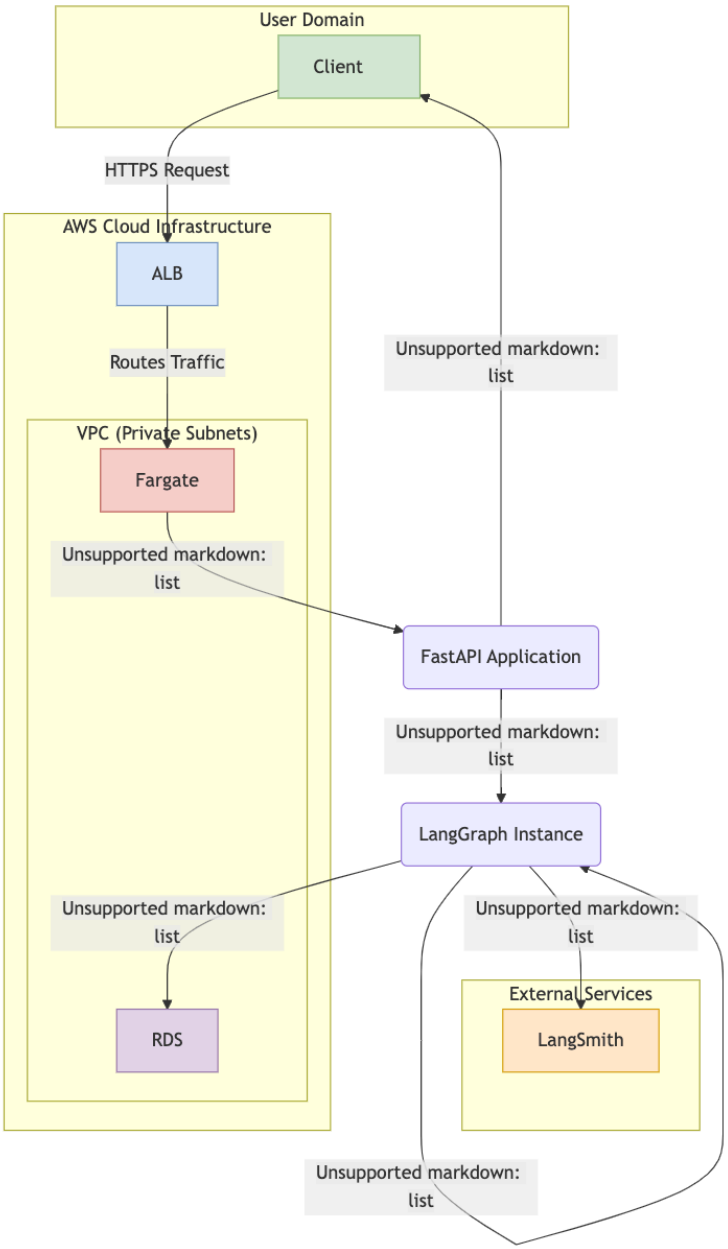
1. エグゼクティブサマリーとシステムアーキテクチャ

1.1. システム概要

本仕様書は、複雑な推論、ツール利用、そして人間の監督を必要とするタスクのために設計された、本番環境に対応可能なマルチエージェントAIシステムのアーキテクチャと実装について詳述する。
このシステムの核となるのは、LangGraphを用いて構築されたエージェントワークフローであり、これにより明示的な制御と状態管理が可能となる 。重要な機能として人間参加型（Human-in-the-Loop, HITL）能力が挙げられる。これは、人間の検証や修正のためにエージェントのタスクを中断・再開することを可能にし、クリティカルな操作における信頼性と安全性を確保する 。
システムはFastAPIバックエンドを介して公開され、サーバーセントイベント（Server-Sent Events, SSE）を活用してエージェントの推論プロセスをリアルタイムかつ透過的にエンドユーザーにストリーミングする。デプロイメントは、AWS ECS Fargate上のTerraform定義インフラストラクチャと、安全なGitHub Actions CI/CDパイプラインを用いて完全に自動化される。

1.2. アーキテクチャ設計図

本システムは、クライアントからのリクエスト受付から、AWS上のインフラでのエージェント実行、そしてリアルタイムなレスポンスストリーミングまで、一貫したフローで構成される。全体のアーキテクチャは以下の通りである。



フローの説明:

1. **クライアント**は、FastAPIアプリケーションへのHTTPリクエストを開始する。
2. **AWS Application Load Balancer (ALB)** がTLSを終端し、リクエストをECS Fargate上で稼働するFastAPIサービスにルーティングする。
3. **FastAPIアプリケーション**はリクエストを受け取り、LangGraphエージェントの実行を開始する。主に2つのエンドポイントを持つ。
    - /stream：ユーザーのクエリを受け付け、新しいLangGraphスレッドを開始し、エージェントの実行イベントをSSEでストリーミングする。
    - /resume：人間のフィードバック（承認や修正データ）を受け取り、中断されたグラフの状態を更新して実行を再開する。
4. **LangGraphグラフ**は、FastAPIプロセス内でエージェントのロジックを実行する。ツールを使用したり、LLMを呼び出したり、状態に基づいて次のアクションを決定したりす

る。
5. グラフの各ステップの状態は、**AWS RDS for PostgreSQL**に永続化される。これにより、長時間のタスクやHITLの中断・再開が可能になる。
6. エージェントの実行トレースは、観測性とデバッグのために**LangSmith**プラットフォームに送信される。

1.3. フレームワーク選定理由：LangGraph

エージェントシステムのフレームワーク選定は、システムの制御性、拡張性、および本番環境での信頼性を決定する上で最も重要なアーキテクチャ上の決定である。AutoGen、CrewAI、LangGraphの3つの主要なフレームワークを比較検討した結果、本システムの要件にはLangGraphが最も適していると結論付けた。

1.3.1. 明示的な制御の必要性

AutoGenやCrewAIのようなフレームワークは強力な抽象化を提供するが、エージェント間のインタラクションを自由形式の会話（AutoGen）や役割ベースの順次的なタスク引き渡し（CrewAI）としてモデル化する傾向がある 。これは、本番システムで要求される複雑で非線形、あるいは周期的なワークフローを管理する上で、予測不可能性や困難さを生じさせる可能性がある 。特に、エージェントが意図しないループに陥ったり、タスクから逸脱したりするリスクを制御することが難しい。
対照的に、LangGraphはエージェントワークフローを明示的なステートマシンまたはグラフ（StateGraph）として扱う 。ノードが関数（エージェント、ツール）を表し、エッジが遷移を定義する。このアプローチは、システムの振る舞いを決定論的かつデバッグ可能にするための、きめ細かな制御を提供する 。

1.3.2. 優れた状態管理と永続性

LangGraphの中核概念は、グラフを通じて渡される、可変（mutable）かつ永続的な状態オブジェクトである 。これは、そのチェックポインターシステムと組み合わせることで、堅牢なHITL、長時間実行タスク、そしてフォールトトレランスを可能にする基盤技術となる。これらの機能は、他のフレームワークでは明示的にサポートされていないか、実装がより困難である。

1.3.3. シームレスな人間参加型（HITL）統合

LangGraphは、人間とエージェントの協調作業を念頭に置いて設計されている。interruptメカニズムは後付けの機能ではなく、第一級の機能として提供されており、任意のノードでワークフローを一時停止し、人間の入力を待ってから再開することができる 。これは、人間の承認や修正を必要とする本システムのクリティカルな要件である。

1.3.4. エージェントフレームワークの成熟度

エージェントフレームワークの動向を分析すると、明確な成熟の過程が見て取れる。初期のフレームワークは、自律性と迅速なプロトタイピングを優先していた（例：CrewAIの役割ベースのアプローチ ）。しかし、これらのシステムが本番環境へと移行するにつれて、開発者は予測不能な振る舞い、フィードバックループの不安定性、デバッグの困難さといった課題に直面した 。

この経験から、明示的な状態、明確な制御フロー、堅牢なエラーハンドリングといった、従来のソフトウェア工学に近い特性を持つシステムへの需要が高まった。LangGraphの登場は、この市場の成熟を象徴している。それは単なる「箱入りエージェント」ではなく、オーケストレーションライブラリとしての側面が強い。これは、本番用のエージェントワークフローを、単なるプロンプトの問題としてではなく、ソフトウェアエンジニアリングの問題として扱う必要があるという認識を反映している。したがって、LangGraphの選択は単なる技術的な好みではなく、信頼性が高く、管理可能でデバッグ可能なAIシステムを構築するという、より広範な業界のトレンドに合致するものである。

特徴	LangGraph	AutoGen	CrewAI
アーキテクチャ	グラフベースのステートマシン	階層的なイベント駆動型会話	役割ベースのタスク指向クルー
状態管理	明示的、永続的な共有状態オブジェクト	各エージェント内の暗黙的な会話履歴	タスク間の構造化された出力の引き渡し
エージェント間通信	共有状態を介した間接的な通信	非同期メッセージパッシング	構造化されたタスクの順次引き渡し
制御フロー	明示的なエッジと条件分岐による完全な制御	会話パターンに基づく動的なフロー	事前定義されたプロセス（順次/並列）
HITL適合性	非常に高い（interrupt機能が組み込み）	中程度（実行中の介入が可能）	限定的（カスタムフローの実装が必要）

2. 開発環境とコンテナ化

2.1. Poetryによる依存関係管理

本プロジェクトでは、Pythonの依存関係管理ツールとしてPoetryを採用する。Poetryは、従来のrequirements.txtファイルと比較して、堅牢な依存関係解決、ロックファイル（poetry.lock）の生成、本番用と開発用の依存関係の明確な分離といった点で優れている。

pyproject.toml 設定

以下に、本プロジェクトの完全なpyproject.tomlファイルを示す。

```
[tool.poetry]
name = "hitl-agent-system"
version = "0.1.0"
description = "A Human-in-the-Loop AI agent system using LangGraph and
FastAPI."
authors =
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.11"
fastapi = "^0.111.0"
uvicorn = {extras = ["standard"], version = "^0.30.1"}
langgraph = "^0.1.1"
langchain-openai = "^0.1.8"
sqlalchemy = "^2.0.31"
```

```
asyncpg = "^0.29.0"
sse-starlette = "^2.1.0"
langgraph-checkpoint-sqlite = "^0.1.1"
langgraph-checkpoint-postgres = "^0.1.1"
pydantic-settings = "^2.3.4"
python-dotenv = "^1.0.1"
psycopg2-binary = "^2.9.9" # For synchronous parts of
Alembic/SQLAlchemy if needed

[tool.poetry.group.dev.dependencies]
pytest = "^8.2.2"
httpx = "^0.27.0"
black = "^24.4.2"
ipykernel = "^6.29.4" # For notebook-based testing

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

- **[tool.poetry.dependencies]**: 本番環境でアプリケーションを実行するために必要なライブラリを定義する。langgraph-checkpoint-sqliteはローカル開発での状態永続化に、langgraph-checkpoint-postgresは本番環境のRDSとの連携に使用する。
- **[tool.poetry.group.dev.dependencies]**: テスト、リンティング、ローカルでのデバッグなど、開発時にのみ必要なライブラリを定義する。これにより、本番用のコンテナイメージに不要なパッケージが含まれるのを防ぐ。

2.2. 本番環境対応のDocker化

最小限で安全な最終イメージを作成するために、単一のDockerfile内で複数のステージを利用するマルチステージビルド戦略を採用する。これは、単純なシングルステージビルドと比較して、攻撃対象領域とイメージサイズを大幅に削減する重要な最適化である。

Dockerfile

```
# Stage 1: Builder - Installs dependencies
FROM python:3.11-slim-bullseye AS builder

# Set environment variables for Poetry
ENV POETRY_HOME="/opt/poetry"
ENV POETRY_VIRTUALENVS_CREATE=true
ENV POETRY_VIRTUALENVS_IN_PROJECT=true
ENV PATH="$POETRY_HOME/bin:$PATH"

# Install Poetry
RUN apt-get update && apt-get install -y curl && \
    curl -sSL https://install.python-poetry.org | python3 -

# Copy only dependency-defining files to leverage Docker cache
```

```
WORKDIR /app
COPY pyproject.toml poetry.lock./

# Install production dependencies
RUN poetry install --no-root --without dev

# -----------------------------------------------------------------

# Stage 2: Production - Creates the final, lean image
FROM python:3.11-slim-bullseye AS production

# Create a non-root user for security
RUN useradd --create-home appuser
USER appuser
WORKDIR /home/appuser/app

# Copy virtual environment from builder stage
COPY --from=builder /app/.venv./.venv
ENV PATH="/home/appuser/app/.venv/bin:$PATH"

# Copy application source code
COPY./app./app

# Expose port and define the command to run the application
EXPOSE 8000
CMD
```

- **builderステージ**:
    1. 公式のPythonスリムイメージから開始する。
    2. Poetryをインストールする。
    3. pyproject.tomlとpoetry.lockのみをコピーする。これにより、ソースコードが変更されても依存関係のレイヤーはキャッシュされ、ビルドが高速化される。
    4. poetry install --no-root --without devを実行し、本番用の依存関係のみを仮想環境にインストールする。
- **productionステージ**:
    1. 同じくスリムなベースイメージから開始する。
    2. セキュリティ向上のため、非ルートユーザー（appuser）を作成し、使用する。
    3. builderステージから仮想環境をコピーする。これにより、最終イメージにPoetry自体を含める必要がなくなる。
    4. アプリケーションのソースコードをコピーする。
    5. 本番環境で推奨されるgunicornとuvicornワーカーを使用してアプリケーションを実行するCMDを定義する。

2.3. ローカル開発環境のセットアップ

ローカル環境のオーケストレーションにはdocker-compose.ymlを使用する。これにより、本番のコンテナ化された環境を密接に模倣しつつ、ライブコード更新の利便性を提供する。
コンテナ化のプロセスは、開発者体験（ホットリロード、容易なデバッグ）と本番環境の要件

（最小サイズ、セキュリティ、不変性）という、しばしば相反する2つの目標を両立させる必要がある。単純なアプローチでは、両方に単一のDockerfileを使用し、結果として肥大化した本番イメージや扱いにくいローカル設定になりがちである。本仕様書で採用する専門的なアプローチは、構造化された単一のマルチステージDockerfileを使用し、docker-compose.ymlを介してローカル開発用にその実行を適応させる（例：ボリュームのマウントやCMDの上書き）。これは、ソフトウェア開発ライフサイクル全体への成熟した理解を示すものである。

docker-compose.yml

```
version: '3.8'

services:
  api:
    build:
      context:.
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      -./app:/home/appuser/app/app
      -./checkpoints.sqlite:/home/appuser/app/checkpoints.sqlite #
Persist SQLite DB
    env_file:
      -.env
    command: ["uvicorn", "app.main:app", "--host", "0.0.0.0",
"--port", "8000", "--reload"]
```

- **apiサービス**:
  - DockerfileからFastAPIアプリケーションをビルドする。
  - ソースコードをボリュームとしてマウントし、ホットリロードを可能にする。
  - ローカルでの状態永続化のために、SQLiteデータベースファイル（checkpoints.sqlite）をマウントする。
  - .envファイルから環境変数を読み込む。
  - commandを上書きし、開発用のuvicornサーバーを--reloadフラグ付きで実行する。

3. コアエージェントワークフロー：LangGraph実装

3.1. 状態スキーマの定義

グラフの心臓部はその状態である。ノード間で渡される中央の状態オブジェクトとして、AgentStateという名前のTypedDictクラスを定義する 。

```
from typing import TypedDict, Annotated, List
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages

class AgentState(TypedDict):
    """
    エージェントのワークフロー全体で情報を保持する状態オブジェクト。
```

```
    Attributes:
        messages: 会話の全履歴。add_messagesアノテーションにより、
                  新しいメッセージは上書きではなく追記される。
        input_query: ユーザーからの最初のクエリ。
        tool_outputs: ツール実行の結果を格納するリスト。
        human_in_loop: プロセスが人間の介入を待って一時停止しているかを示すフラグ。
    """
    messages: Annotated, add_messages]
    input_query: str
    tool_outputs: list
    human_in_loop: bool
```

- **messages: Annotated[list, add_messages]**: この特殊なアノテーションは、LangGraphが新しいメッセージをリストに追記するように指示する。これにより、上書きされることなく会話履歴が自然に蓄積される 。

3.2. ノードの実装

ワークフローの各ステップはノードとして実装される。

- **agentノード**: 主要な推論ノード。現在のAgentStateを受け取り、messagesをプロンプトに整形し、ツールがバインドされたLLMを呼び出す。出力はAIMessageであり、ユーザーへの応答またはツール呼び出しのリクエストを含む 。
- **tool_node**: langgraph.prebuiltからToolNodeを利用する。このノードは、agentノードによって要求されたツール呼び出しを効率的に実行し、その結果をToolMessageオブジェクトとして状態に追記する 。
- **human_reviewノード**: このカスタムノードは、呼び出されるとinterrupt()を返す。これがグラフの実行を一時停止させる主要なメカニズムである。人間の承認が必要な状態の遷移先となる 。

3.3. 条件付きエッジによる制御フロー

エージェントのロジックはエッジによって定義される。add_conditional_edgesを使用して、動的で周期的なフローを作成する 。

```
graph TD
    A --> B(agent);
    B --> C{should_continue};
    C -- tool_calls --> D[tool_node];
    C -- requires_approval --> E[human_review];
    C -- continue --> F;
    D --> B;
    E --> G((PAUSED));

    style B fill:#F8CECC,stroke:#B85450
    style D fill:#DAE8FC,stroke:#6C8EBF
    style E fill:#FFE6CC,stroke:#D79B00
```

- **エントリーポイント**: グラフはagentノードから開始する。
- **ルーティング関数 (should_continue)**: agentノードの実行後、この関数が状態内の最新のメッセージを検査する。
    1. メッセージにツール呼び出しが含まれている場合、"tools"を返し、tool_nodeにルーティングする。
    2. エージェントの応答に特定のキーワード（例：[requires_approval]）が含まれている場合、"human_review"を返し、HITLノードにルーティングする。
    3. それ以外の場合、ENDを返し、実行を終了する。
- tool_nodeの実行後は、標準のadd_edgeが制御をagentノードに戻し、中心的な推論ループを形成する。

3.4. チェックポインターによる永続化

LangGraphのチェックポインターシステムは、単なる永続化機能以上の役割を果たす。それは、HITL、フォールトトレランス、時間遡行デバッグといった、システムの最も高度な能力を可能にするアーキテクチャ上の要となる要素である 。interrupt機能は、エージェントの状態が数秒から数時間にわたって完全に保存されて初めて実用的になる。チェックポインターはこの耐久性のある状態スナップショットを提供する。このため、チェックポインターデータベースはミッションクリティカルなコンポーネントとして扱い、本番環境ではAWS RDSのようなマネージドサービスの利用を正当化する。

- **ローカル開発 (AsyncSqliteSaver)**: セットアップを容易にするため、ローカル環境ではlanggraph_checkpoint_sqliteを使用する。AsyncSqliteSaverをローカルファイル（checkpoints.sqlite）で初期化し、graph.compile()メソッドに渡す方法を示す 。これにより、別のデータベースサーバーを必要とせずに、開発中に完全な状態永続性が提供される。
- **本番環境 (AsyncPostgresSaver)**: SQLiteは書き込み競合のため本番環境には適していないことを明記する 。langgraph-checkpoint-postgresパッケージのAsyncPostgresSaverへの移行パスを定義する。接続詳細は環境変数を通じて管理され、Terraformで管理されるRDSインスタンスから供給される。
- **thread_idの役割**: 並行する会話を管理するためのthread_id（configurable辞書で渡される）の重要性を説明する。各thread_idは、チェックポインターデータベース内で一意の永続的な状態履歴を表す 。

4. APIレイヤーとFastAPIによるリアルタイムストリーミング

4.1. アプリケーションのセットアップとライフスパン管理

FastAPIアプリケーションは、ルーター、サービス、モデルなどの関心事を明確に分離して構成する。アプリケーションの起動・終了イベントの管理には、lifespan非同期コンテキストマネージャーを使用する。

- **起動時**: AsyncPostgresSaverチェックポインターを初期化し、LangGraphのgraphオブジェクトをコンパイルする。これらはアプリケーションの状態（例：app.state.graph）に保存され、リクエストハンドラからアクセス可能になる。
- **終了時**: データベース接続を適切に閉じる。

4.2. ストリーミングエンドポイント (SSE)

エージェントシステムの「ブラックボックス」的な性質は、特に長時間実行されるタスクにお

いてユーザー体験を損なう。LangGraphのastream_eventsはエージェントの内部状態変化、ツール呼び出し、推論ステップの豊富なストリームを提供する。この情報をクライアントにプッシュするという要件に対し、SSEの単方向性（サーバーからクライアント）は技術的に完全に一致する。これは、この特定のユースケースにおいて、双方向通信のオーバーヘッドを持つWebSocketよりも実装がシンプルでスケールしやすい。したがって、SSEの選択は、LangGraphの観測可能性と最適なフロントエンド通信プロトコルを結びつけ、透過的でインタラクティブなユーザー体験を創出するための意図的な設計判断である。

- **エンドポイント**: POST /stream
- **機能**: このエンドポイントはユーザーのクエリとthread_idを受け付ける。media_type="text/event-stream"を指定したStreamingResponseを使用してSSE接続を確立する。
- **ジェネレーター関数**: StreamingResponseには非同期ジェネレーター関数からデータが供給される。この関数は、コンパイル済みのLangGraphオブジェクトに対してgraph.astream_events(...)を呼び出し、ユーザー入力とthread_idを渡す。
- **イベントストリーミング**: ジェネレーターのasync forループ内で、astream_eventsから受け取った各イベント（ノードの開始、終了、ツール出力など）はSSEメッセージ（data: {json.dumps(event)}\n\n）としてフォーマットされ、クライアントにyieldされる。これにより、エージェントの実行状況がライブで詳細に表示される。

4.3. 人間介入エンドポイント

- **エンドポイント**: POST /resume
- **機能**: グラフが中断され、人間のフィードバックを待っている状態の後に呼び出される。
- **リクエストボディ**: 中断されたグラフのthread_idと、人間が提供したデータ（承認メッセージや修正情報など）を受け付ける。
- **ロジック**:
    1. graph.update_state()を使用して、指定されたthread_idのグラフ状態に人間のフィードバックを注入する 。
    2. graph.astream_events(None, config={"configurable": {"thread_id": thread_id}})を呼び出し、中断された時点からグラフの実行を再開する。この応答も、必要であれば新しいSSE接続を介してクライアントにストリーミングすることができる。

4.4. 非同期データベース統合

- asyncpgドライバーと共に、SQLAlchemy 2.0の非同期機能を使用する。
- 各リクエストのAsyncSessionのライフサイクルを管理するために、依存性注入関数（get_db_session）を作成し、接続が適切に開閉されることを保証する。
- この非同期セッションはAsyncPostgresSaverチェックポインターによって使用され、状態データベースとのすべてのやり取りがノンブロッキングであり、FastAPIの非同期性と互換性があることを保証する。

5. Terraformによる本番インフラストラクチャ (IaC)

このセクションでは、完全でモジュール化されたTerraformファイル群を提供する。

5.1. ネットワーク基盤 (VPC & Subnets)

- vpc.tf: 高可用性のために2つのアベイラビリティゾーンにまたがるaws_vpc、パブリック

およびプライベートaws_subnet、aws_internet_gateway、そしてプライベートサブネットからのアウトバウンドトラフィック用のaws_nat_gatewayを定義する。

5.2. データベース層 (AWS RDS)

- rds.tf: PostgreSQL用のaws_db_instanceまたはaws_rds_clusterを定義する。インスタンスサイズ、ストレージ、フェイルオーバーのためのマルチAZデプロイメントを指定する。
- RDSインスタンス用のaws_security_groupは、ECS Fargateサービスのセキュリティグループからのみポート5432へのインバウンドトラフィックを許可するように設定する。
- データベースの認証情報はaws_secretsmanager_secretを介して管理する。

5.3. コンテナオーケストレーション (AWS ECS Fargate)

- ecr.tf: Dockerイメージを保存するためのaws_ecr_repositoryを定義する。
- ecs.tf:
    - aws_ecs_cluster: サービスのための論理的なグループ。
    - ECSタスク実行用のaws_iam_role: ECRからのプルとCloudWatch Logsへの書き込み権限を付与する。
    - aws_ecs_task_definition: コンテナイメージ、CPU/メモリ割り当て、ポートマッピング、環境変数（データベースシークレットのARNを含む）を指定する。
    - aws_ecs_service: 実行中のタスクを管理し、ALBターゲットグループにリンクし、希望するタスク数を定義し、Fargateの起動タイプとネットワーク構成（プライベートサブネットとセキュリティグループ）を指定する。

5.4. トラフィック管理 (Application Load Balancer)

- alb.tf: aws_lb、aws_lb_target_group、aws_lb_listenerを定義する。リスナーはポート443でHTTPSトラフィックを受け付け、ECSターゲットグループに転送するように設定する。

Terraformモジュール出力と説明

インフラプロビジョニングのステップとCI/CDデプロイメントのステップ間の契約として、Terraformの出力は運用上不可欠である。例えば、ecr_repository_urlはDockerイメージをタグ付けしてプッシュするためにGitHub Actionsワークフローで必要となり、alb_dns_nameはユーザーがアクセスする最終的なエンドポイントとなる。これらを明確に文書化することで、システムの理解と管理が容易になる。

出力名	説明	値の例
alb_dns_name	アプリケーションロードバランサーの公開DNS名	hitl-agent-alb-1234567890.us-east-1.elb.amazonaws.com
ecr_repository_url	DockerイメージをプッシュするためのECRリポジトリURL	123456789012.dkr.ecr.us-east-1.amazonaws.com/hitl-agent-repo
rds_endpoint	RDS PostgreSQLデータベースの接続エンドポイント	hitl-agent-db.c123xyz.us-east-1.rds.amazonaws.com
ecs_cluster_name	ECSクラスターの名前	hitl-agent-cluster

6. GitHub ActionsによるCI/CD自動化

6.1. OIDCによる安全なAWS認証

本番環境のCI/CDパイプラインでは、長期的なAWS_ACCESS_KEY_IDやAWS_SECRET_ACCESS_KEYをGitHubのシークレットに保存する手法を明確に避ける。代わりに、OpenID Connect (OIDC) を利用して、安全で一時的な、クレデンシャルレスの認証メカニズムを構築する。

- **AWSセットアップ**: AWSのIAMコンソールで、token.actions.githubusercontent.comを指すIAM OIDC IDプロバイダーを作成する。
- **IAMロール**: GitHub OIDCプロバイダーが引き受けることを許可する信頼ポリシーを持つIAMロールを作成する。このポリシーは、特定のリポジトリとブランチ（例：repo:my-org/my-repo:ref:refs/heads/main）に限定する条件付きとする。
- **GitHubワークフロー設定**: ワークフローファイルにid-tokenを要求するためのpermissionsブロックを含め、aws-actions/configure-aws-credentialsアクションにIAMロールのARNを指定する。

6.2. ビルド＆プッシュワークフロー (build-and-push.yml)

多くのCI/CDの例では、アーティファクトのビルドとインフラのデプロイを単一のワークフローにまとめている。しかし、本番グレードの堅牢なパターンは、これらを分離することである。ビルドプロセスとデプロイプロセスは論理的に異なる活動であり、異なる失敗モードとトリガーを持つ。これらを組み合わせると、Terraformの適用失敗がビルドプロセス全体の再実行を要求するなど非効率的になる。
本仕様書では、この分離パターンを実装する。「ビルド」ワークフローの唯一の責任は、バージョン管理された不変のアーティファクト（GitのSHAでタグ付けされたDockerイメージ）を生成し、レジストリ（ECR）に保存することである。「デプロイ」ワークフローは、このアーティファクトの正常な作成によってトリガーされ、その特定のバージョンを使用して実行中のインフラを更新する。この分離は、関心事の分離を改善し、効率を高め、将来的にブルーグリーンデプロイメントやロールバックなどの高度なデプロイ戦略を可能にする。

```
name: Build and Push Docker Image

on:
  push:
    branches:
      - main

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      id-token: write
      contents: read

    steps:
      - name: Checkout repository
```

```
        uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::${{ secrets.AWS_ACCOUNT_ID
}}:role/GitHubAction-OIDC-Role
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v2

      - name: Build, tag, and push image to Amazon ECR
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          ECR_REPOSITORY: ${{ secrets.ECR_REPOSITORY_NAME }}
          IMAGE_TAG: ${{ github.sha }}
        run: |
          docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG.
          docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
```

- **トリガー**: mainブランチへのプッシュ。
- **ステップ**: コードのチェックアウト、OIDCによるAWS認証、ECRへのログイン、Dockerイメージのビルド、GitのSHAをタグとしてイメージをECRリポジトリにプッシュする。

6.3. デプロイワークフロー (deploy.yml)

```
name: Deploy to ECS

on:
  workflow_run:
    workflows:
    types:
      - completed

jobs:
  deploy:
    runs-on: ubuntu-latest
    if: ${{ github.event.workflow_run.conclusion == 'success' }}
    permissions:
      id-token: write
      contents: read

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
```

```
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::${{ secrets.AWS_ACCOUNT_ID
}}:role/GitHubAction-OIDC-Role
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v3

      - name: Terraform Init
        run: terraform init
        working-directory:./terraform

      - name: Terraform Apply
        run: |
          terraform apply -auto-approve \
            -var="docker_image_tag=${{
github.event.workflow_run.head_sha }}"
        working-directory:./terraform
```

- **トリガー**:「Build and Push Docker Image」ワークフローの正常完了。
- **ステップ**: コードのチェックアウト、OIDCによるAWS認証、Terraformのセットアップ、terraform init、terraform plan、terraform applyの実行。新しいイメージをデプロイするために、ECRイメージタグ（GitのSHA）がTerraformに変装として渡される。

7. 運用準備とセキュリティに関する考慮事項

7.1. 設定とシークレットの管理

ECSタスク定義のTerraform設定には、平文のシークレットを含めない。代わりに、AWS Secrets Managerに保存されたシークレット（RDSパスワードなど）のARNを参照する。ECSタスク実行ロールには、このシークレットを取得する権限が付与される。これにより、インフラストラクチャコードと機密データが安全に分離される。

7.2. ロギング、モニタリング、観測可能性

- **LangSmith**: エージェントの実行を詳細にトレースするために、アプリケーションコードはLangSmithのAPIキー（環境変数として渡される）で設定される。これはエージェントの振る舞いをデバッグするために任意ではなく、不可欠であることを強調する 。
- **AWS CloudWatch**: Fargateタスクは、コンテナのログ（stdout/stderr）をCloudWatch Log Groupにストリーミングするように設定される。これは、アプリケーションの健全性の監視とインフラレベルの問題のデバッグに不可欠である。

7.3. アプリケーションセキュリティの強化

セキュリティは後付けで考慮されるべきではない。OWASP Application Security Verification Standard (ASVS)のような認識された標準にマッピングされたコンプライアンスチェックリス

トを明示的に含めることで、本仕様書は機能的な設計から本番準備の整った設計へと昇華する。これにより、主要なセキュリティ管理策の検討が強制され、検証のための明確で実行可能なリストが提供される。

ASVS ID	管理策の説明	実装状況/メモ
V14.1 ビルドとデプロイ		
14.1.1	安全で再現可能なビルドとデプロイのプロセスが自動化されていることを確認する。	GitHub ActionsとTerraformにより実装済み。
V14.3 意図しないセキュリティ情報開示		
14.3.3	HTTPレスポンスがシステムコンポーネントの詳細なバージョン情報を公開しないことを確認する。	Gunicorn/UvicornはServerヘッダーを抑制するように設定する。
V14.4 HTTPセキュリティヘッダー		
14.4.3	XSS攻撃の影響を緩和するために、Content Security Policy (CSP) レスポンスヘッダーが設定されていることを確認する。	このAPIのみのサービスには直接適用されないが、フロントエンドを追加する場合は必須。
14.4.5	すべてのレスポンスとサブドメインにStrict-Transport-Securityヘッダーが含まれていることを確認する。	Application Load Balancerで設定済み。
14.4.7	Webアプリケーションのコンテンツが第三者のサイトに埋め込まれないように、frame-ancestorsやX-Frame-Optionsヘッダーが適切に使用されていることを確認する。	APIエンドポイントには直接関連しないが、ベストプラクティスとしてALBでX-Frame-Options: DENYを設定する。

引用文献

1. LangGraph - LangChain, https://www.langchain.com/langgraph
2. Conceptual Guides - LangGraph, https://www.baihezi.com/mirrors/langgraph/concepts/index.html
3.
4. Add human-in-the-loop, https://langchain-ai.github.io/langgraph/tutorials/get-started/4-human-in-the-loop/ 4. Technical Comparison of AutoGen, CrewAI, LangGraph, and ..., https://ai.plainenglish.io/technical-comparison-of-autogen-crewai-langgraph-and-openai-swarm-1e4e9571d725
5. Position: Towards a Responsible LLM-empowered Multi-Agent Systems - arXiv, https://arxiv.org/html/2502.01714v1

6. Why Do Multi-Agent LLM Systems Fail? - arXiv, https://arxiv.org/pdf/2503.13657?
7. LangGraph - LangChain Blog, https://blog.langchain.com/langgraph/
8. Comparing LLM Agent Frameworks Controllability and ..., https://scalexi.medium.com/comparing-llm-agent-frameworks-langgraph-vs-autogen-vs-crew-ai-part-i-92234321eb6b
9. LangGraph persistence - GitHub Pages, https://langchain-ai.github.io/langgraph/concepts/persistence/
10. Introduction - CrewAI, https://docs.crewai.com/
11. 11. Nodes, Edges, States & Graph in LangGraph — Basics | by ..., https://medium.com/@official.hardcodeconcepts/nodes-edges-states-graph-in-langgraph-basics-3bdc7e9954b6
12. Customize state - Docs by LangChain, https://docs.langchain.com/langgraph-platform/langgraph-basics/5-customize-state
13. Call tools - GitHub Pages, https://langchain-ai.github.io/langgraph/how-tos/tool-calling/
14. How can I implement conditional edges in Langgraph for agent decision? - Stack Overflow, https://stackoverflow.com/questions/79433194/how-can-i-implement-conditional-edges-in-langgraph-for-agent-decision
15. langchain-ai/langgraph: Build resilient language agents as graphs. - GitHub, https://github.com/langchain-ai/langgraph
16. 16. langgraph-checkpoint-sqlite - PyPI, https://pypi.org/project/langgraph-checkpoint-sqlite/
17. Simple LangGraph Implementation with Memory AsyncSqliteSaver Checkpointer — FastAPI, https://medium.com/@devwithll/simple-langgraph-implementation-with-memory-asyncsqlitesaver-checkpointer-fastapi-54f4e4879a2e
18. from langgraph.checkpoint.sqlite import SqliteSaver was wrong #3707 - GitHub, https://github.com/langchain-ai/langgraph/discussions/3707
19. README.md - langchain-ai/langgraph - GitHub, https://github.com/langchain-ai/langgraph/blob/main/README.md